2INC0 summary - Samenvatting Operating systems

Operating systems (Technische Universiteit Eindhoven)

Midterm notes

Contract
Structure:
Short description
@param
@modifies {}
@pre {}
@post {@code a (\forall i; a.has(i); a[i] == **...**}
@throws ***Exception*** when {}
void method(***params***) throws ***Exception*** {
}

e.g.

```
/*
 * Increments all array elements whose index occurs in the given set,
 *   by the maximum value in the array.
 *
 * @param a  the given array to be updated
 * @param s  the given set
 * @pre {@code a != null && s != null &&
 *        (\forall i; s.contains(i); a.has(i))}
 * @modifies a
 * @throws NullPointerException  if {@code a == null || s == null}
 * @throws IllegalArgumentException
 *        if {@code ! (\forall i; s.contains(i); a.has(i))}
 * @post {@code (\forall i; s.contains(i); a[i] == \old(a[i]) + M)}
 *     where {@code M == \old((\max i; a.has(i); a[i]))}
 */
void incSelectionByMax(int[] a, Set s)
       throws NullPointerException, IllegalArgumentException
{ /* ... */ }
```

Robustness
Definition:
A method that always throws an exception when its precondition is violated is said to be robust.

Advantage of robustness:
Makes it clear that there is not a problem with the method, however with the input delivered to it

Disadvantages of robustness:
1. Overhead to write the precondition checks and to test them
2. More code to read and understand the function
3. Runtime overhead in checking the precondition

4. Runtime overhead in catching exceptions

When to strive for robustness
It is especially important to strive for robustness of method on public interfaces, however, if precondition checking is too costly, then it is not a good idea.

Exceptions
Definition:
Thrown when the precondition of a function is violated.

NullPointerException: trying to access a object that is null
IllegalArgumentException: precondition doesn't hold

Advantages of exceptions:
1. Exceptions can give extra information
2. Exceptions cannot be ignored easily

Disadvantages of exceptions:
Same as robustness

Testing
What to test:
1. Normal input
2. Boundary (e.g. empty array)
3. Post condition already holds
4. Exceptions

JavaDoc Implementation
```
public void testMethodToTest() {
  try {
    int valueWithError = 3;
    methodToTest(valueWithError);
    fail("Expected methodToTest to throw IllegalArgumentException");
  } catch (IllegalArgumentException e) {
    // Expected exception: Test passes!
    return;
  } catch (Exception e) {
    // Unexpected exception: Test failed!
    fail("Expected IllegalArgumentException, got " + e.getClass().getName());
  }
}
```

Functional decomposition
Definition: breaking a function into smaller, independent functions to simplify complexity and facilitate testing

Modularity
Definition: the degree to which a system's components can be separated and recombined

Polymorphism
the ability of different objects to respond in their own way to the same message or method call

UML

Name	Notation	Description
Class	A – a1: T1 – a2: T2 + o1(): void + o2(): void	Description of the structure and behavior of a set of objects
Abstract class	A oder {abstract} A	Class that cannot be instantiated
Association	A — B A ↔ B A ×→ B	Relationship between classes: navigability unspecified, navigable in both directions, not navigable in one direction

Name	Notation	Description
n-ary association	A ◇ B C	Relationship between n (here 3) classes
Association class	A — B C	More detailed description of an association
xor relationship	B {xor} C A	An object of **C** is in a relationship with an object of **A** or with an object of **B** but not with both

Name	Notation	Description
Shared aggregation	A ◇ B	Parts-whole relationship (A is part of B)
Strong aggregation = composition	A ◆ B	Existence-dependent parts-whole relationship (A is part of B)
Generalization	A ▷ B	Inheritance relationship (A inherits from B)
Object	o:C	Instance of a class
Link	o1 — o2	Relationship between objects

Abstract Data Type (ADT)
A data type that provides a set of abstract values and operations while having an encapsulated implementation.

Encapsulation: the internal details of the data structure are hidden from the user, who interacts with the data structure through a set of well-defined operations

Design Patterns

- Singleton pattern: When there should be exactly one instance of a class in the entire application
- Strategy pattern: When you have multiple algorithms for a specific task and you want to decide the algorithm to use at runtime
- Template method pattern: When there's a skeleton of an algorithm with multiple steps, and you want to allow subclasses to redefine certain steps of the algorithm without changing its structure
- Factory method pattern: When you want to provide a creation interface for creating objects but allow subclasses to alter the type of objects that will be created.
- Iteration design pattern: When you need to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representatio
- Listener design pattern: allows objects (listeners) to 'listen' and respond to events generated by other objects
- Observer design pattern: Used when a change in one object's state requires changes in other dependent objects

Singleton pattern

```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Initialization code here
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    ... other methods ...
}
```

Strategy pattern

```java
abstract class Strategy {
    void execute();
}

class ConcreteStrategyA implements Strategy {
    @Override
    public void execute() {
        System.out.println("Executing strategy A");
    }
}

class ConcreteStrategyB implements Strategy {
    @Override
    public void execute() {
        System.out.println("Executing strategy B");
    }
}
```

Template method pattern

```java
abstract class AbstractAlgorithmTemplate {
    public final void executeAlgorithm() {
        step1();
        step2();
    }

    protected abstract void step1();
    protected abstract void step2();
}

class ConcreteAlgorithmA extends AbstractAlgorithmTemplate {
    protected void step1() {
    }

    protected void step2() {
    }

    protected void step3() {
    }
}

class ConcreteAlgorithmB extends AbstractAlgorithmTemplate {
    protected void step1() {
    }

    protected void step2() {
    }

    protected void step3() {
    }
}
```

Factory method pattern

```java
// Classes
abstract class Product {
    abstract void display();
}

class ConcreteProductA extends Product {
    @Override
    void display() {
        System.out.println("Product A");
    }
}

class ConcreteProductB extends Product {
    @Override
    void display() {
        System.out.println("Product B");
    }
}

// Creator using factory method
abstract class Creator {
    protected abstract Product createProduct();

    public void useProduct() {
        Product product = createProduct();
        System.out.print("Using ");
        product.display();
    }
}

class ConcreteCreatorA extends Creator {
    @Override
    protected Product createProduct() {
        return new ConcreteProductA();
    }
}

class ConcreteCreatorB extends Creator {
    @Override
    protected Product createProduct() {
        return new ConcreteProductB();
    }
}
```

Iterator design pattern

```java
// Iterator Interface
public interface Iterator<T> {
    boolean hasNext();
    T next();
}

// Collection Interface
public interface Collection<T> {
    Iterator<T> iterator();
}

// Concrete Collection
class ConcreteCollection<T> implements Collection<T> {
    private T[] items;

    public ConcreteCollection(T[] items) {
        this.items = items;
    }

    @Override
    public Iterator<T> iterator() {
        return new ConcreteIterator();
    }

    private class ConcreteIterator implements Iterator<T> {
        private int currentIndex = 0;

        @Override
        public boolean hasNext() {
            return currentIndex < items.length;
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return items[currentIndex++];
        }
    }
}
```